



UNIVERSITÄT ZU LÜBECK  
INSTITUT FÜR IT-SICHERHEIT

# Ausnutzung spekulativer Ausführung durch drive-by Cache-Angriffe

*Exploiting speculative execution via drive-by cache attacks*

**Masterarbeit**

im Rahmen des Studiengangs  
**Informatik**  
der Universität zu Lübeck

vorgelegt von  
**Moritz Krebbel**

ausgegeben und betreut von  
**Prof. Dr. Thomas Eisenbarth**

mit Unterstützung von  
Ida Bruhns

Lübeck, den 21. Juni 2018



# Inhaltsverzeichnis

<b>1 Grundlagen</b>	<b>1</b>
1.1 Virtuelle Speicherverwaltung . . . . .	1
1.2 Caches . . . . .	1
1.2.1 Assoziativität . . . . .	3
1.2.2 Inclusive und Exclusive . . . . .	3
1.3 Cache-Angriffe . . . . .	4
1.3.1 Flush and Reload . . . . .	4
1.3.2 Prime + Probe . . . . .	4
1.3.3 RSA . . . . .	5
<b>2 Implementierung</b>	<b>7</b>
2.1 Timer in JavaScript . . . . .	7
2.1.1 Verdeckter Kanal . . . . .	9
2.1.2 Angriffe auf RSA Key Generierung . . . . .	10



# 1 Grundlagen

Im folgenden sollen Verfahren und Techniken erläutert werden, welche für das Verständnis der späteren Kapitel essenziell sind.

## 1.1 Virtuelle Speicherverwaltung

Virtuelle Speicherverwaltung stellt eine Abstraktion für die vorhandenen physikalischen Speichermedien wie etwa dem Hauptspeicher oder der Festplatte bereit. Das Betriebssystem übersetzt virtuelle Adressen, welche von Prozessen genutzt werden, mit Hilfe der Hardware in physikalische Adressen. Jedem Prozess steht der gleiche virtuelle Adressraum zur Verfügung, wobei das Betriebssystem dafür Sorge trägt für jeden Prozess die richtige Zuordnung von virtueller zu physikalischer Adresse sicherzustellen. Ein Vorteil der virtuellen Speicherverwaltung ist eine erhöhte Sicherheit durch die Speicherisolierung aller Prozesse. So kann eine fehlerhafte Schreiboperation eines Prozesses keinen Fehler in anderen Prozessen verursachen, da gleiche virtuelle Adressen vom Betriebssystem auf unterschiedliche physikalische Adressen abgebildet werden. Des Weiteren kann ein Prozess mehr Hauptspeicher nutzen als physikalisch vorhanden ist, indem Daten vom Betriebssystem auf andere Speichermedien wie die Festplatte ausgelagert werden. Hiermit werden beispielsweise Anwendungsentwickler entlastet, da sie ihre Software nicht für Systeme mit weniger Hauptspeicher gesondert anpassen müssen.

## 1.2 Caches

Die Geschwindigkeitsentwicklung des Hauptspeichers konnte in den letzten Jahren nicht mit der des Prozessors Schritt halten. Der Cache ist ein im Vergleich zum Hauptspeicher kleinerer, aber schnellerer Pufferspeicher, welcher im aktuellen Kontext häufig benötigte

## 1 Grundlagen

Daten vorhält. Ohne Caches wäre ein Prozessor häufig gezwungen auf Daten des langsamen Hauptspeichers zu warten und würde in seiner Verarbeitungsgeschwindigkeit ausgebremst. Auch in anderen Ebenen sind Caches sinnvoll, wie etwa im Browser, wobei in dieser Arbeit vor allem die Caches im Prozessor von Relevanz sind. Weiter liegt der Fokus der Arbeit auf die im Desktopbereich weit verbreitete x86-Architektur, weshalb mit Intel-Prozessoren der Core-Architektur bestückte Testrechner verwendet werden, da Intels Core-Architektur im x86-Desktopsegment zurzeit den höchsten Marktanteil besitzt [6]. Deshalb werden Erklärungen im Folgenden häufig mit Beispielen der Intel Core-Architektur veranschaulicht.

Ein Cache der Core-Architektur lagert nicht einzelne Bytes, sondern immer gleich 64 Bytes, Cache-Line genannt, auf einmal ein. Dabei werden die 64 Bytes beginnend ab der größten Adresse, welche zugleich kleiner als die Zieladresse und ein Vielfaches von 64 ist, angefragt. Angenommen 4 Bytes Daten an Adresse 0b10110111 werden angefordert, dann lagert der Cache die 64 Bytes beginnend mit der Adresse 0b10000000 ein. Heutige Arbeitsspeichermodule liefern 8 Bytes zeitgleich, wobei die CPU mit einem einzigen Befehl einen Burst von 8 Übertragungen initiieren kann, die das Lesen und Schreiben einer gesamten 64 Byte Cache-Line ermöglichen. Daher ist der Performancenachteil, welcher durch ein Laden von gleich 64 Bytes entsteht, vernachlässigbar und wird von dem Vorteil überwogen, dass die zusätzlich geladenen Bytes mit hoher Wahrscheinlichkeit in den nächsten Zyklen ebenfalls benötigt werden.

Ein Prozessorcaché besteht üblicherweise aus mehreren Ebenen, Cache-Level genannt, wobei die Core-Architektur etwa 3 Cache-Level besitzt, welche absteigend größer und langsamer werden. Ein Intel i7-4770 besitzt pro physischen Kern beispielsweise einen 32 KiB L1-Datencache mit einer Zugriffslatenz von 4-5 Taktzyklen und einen 256 KiB L2-Cache mit einer Latenz von 12 Taktzyklen [4]. Im Unterschied zwischen den beiden ersten Cache-Levels teilen sich in der Core-Architektur alle Kerne den L3-Cache. Dies bedeutet aus Angreifersicht einen großen Vorteil, da ein Programm den Zustand des L3-Caches beeinflusst, unabhängig davon auf welchem Kern es ausgeführt wird. Dagegen muss die Angreiferin beim einem Angriff auf den L1- beziehungsweise L2-Cache sicherstellen, dass ihr Code und das angegriffene Programm auf dem selben physischen Kern ausgeführt

werden.

Die Ersetzungsstrategie legt fest welcher Eintrag aus dem Cache verdrängt wird, sofern alle Einträge des zugehörigen Cache-Sets belegt sind. Intels Core-Prozessoren verdrängen typischerweise den Eintrag, welcher bezogen auf die letzte Zugriffszeit am ältesten ist, auch least-recently-used (LRU) Strategie genannt. Ab der Ivy-Bridge Generation passt Intel diese Strategie situationsbedingt an [8], wobei der Algorithmus hinter den Ersetzungsstrategien ebenfalls nicht öffentlich zugänglich ist.

### 1.2.1 Assoziativität

Sofern die Auswahl des Cache-Eintrags für die Daten einer bestimmten Hauptspeicheradresse keinerlei Beschränkungen unterliegt, wird von einem voll-assoziativen Cache gesprochen. Das andere Extrem wäre ein einfach-assoziativer Cache beziehungsweise eine direkte Abbildung, wobei die Adresse des Hauptspeichers von der die Daten stammen den zu wählenden Cache-Eintrag eindeutig festlegt.

Der L3-Cache eines Intel i7-4770 ist beispielsweise 8 MiB groß und verfügt daher über  $131072(2^{17})$  Cache-Einträge der Größe einer Cache-Line. Wäre dieser Cache nun voll-assoziativ, müsste der Cache bei jeder Anfrage komplett durchsucht werden. Aus diesem Grund ist der Cache in Cache-Sets unterteilt, wobei Daten einer spezifischen Hauptspeicheradresse nur in genau ein Cache-Set eingelagert werden können. Der i7-4770 besitzt 8192 dieser Cache-Sets, womit sich eine Assoziativität von 16 ergibt (Anzahl der Cache-Einträge geteilt durch Anzahl der Cache-Sets), das heißt die Suche nach den Daten einer Hauptspeicheradresse im Cache ist auf 16 Einträge begrenzt. Die Zuordnung von Hauptspeicheradressen zu den Cache-Sets ist nicht öffentlich dokumentiert.

### 1.2.2 Inclusive und Exclusive

Ob die Inhalte eines Caches auch in anderen Cache-Levels verfügbar sind, ist ein für diese Arbeit wichtiges Designkriterium. Ein Cache wird als Inclusive bezeichnet, sofern alle Daten die in einem niedrigen Cache-Level vorliegen zusätzlich auch in den höheren Cache-Levels eingelagert sind. So besitzen die Caches aller Desktop-Versionen der Core-Architektur diese Eigenschaft (Stand Juni 2018). Die Caches der Desktop-Prozessoren des

## 1 Grundlagen

Konkurrenten AMD (beispielsweise Zen-Architektur [2]) sowie der aktuellen Skylake-X Prozessoren [3] für Intels High-Performance Plattform besitzen diese Eigenschaft nicht. Wegen des großen Marktanteils von Intel kann festgehalten werden, dass der Großteil der sich im Einsatz befindlichen Prozessoren mit Inclusive Caches ausgestattet ist.

### 1.3 Cache-Angriffe

Cache-Angriffe beschreiben eine generelle Klasse von Mikro-Architektur Seitenkanalangriffen, welche den Cache, der als geteilte Ressource zwischen verschiedenen Prozessen fungiert, verwenden um Informationen abzugreifen. Durch diesen Angriff können sichere und unsichere Prozesse über den geteilten Cache ungeachtet höher liegender Schutzmechanismen wie virtualisiertem Speicher oder Hypervisor-Systemen kommunizieren. Eine Angreiferin könnte ein Programm entwickeln welches Informationen über den inneren Zustand eines anderen Prozesses sammelt und so AES-Schlüssel [1], RSA-Schlüssel [7], auch über die Grenzen von virtuellen Maschinen hinweg, abgreifen.

#### 1.3.1 Flush and Reload

Ausgang dieses Angriffs ist der x86-Assemblerbefehl `clflush`, welcher eine Adresse entgegennimmt und die dazugehörige Cache-Line invalidiert, sodass die Daten beim nächsten Zugriff aus dem Hauptspeicher geladen werden müssen. TODO

#### 1.3.2 Prime + Probe

Ein Eviction-Set sei eine Menge, welche es vermag einen Cache-Eintrag aus einem Cache zu verdrängen. D.h. ein Eviction-Set welches einen Eintrag aus dem L3-Cache löscht, würde den gleichen Zweck wie der `clflush`-Assemblerbefehl im Flush + Reload Angriff erfüllen. Um einen Eintrag aus dem Cache zu verdrängen, müssen mehrere Adressen der Daten aus dem Eviction-Set von der CPU auf die gleiche Cache-Set wie der zu verdrängende Eintrag abgebildet werden, sodass die Größe eines Eviction-Sets mindestens die Assoziativität des Caches erreichen sollte.

Die Idee beim Prime + Probe Angriff besteht darin, in einer sich wiederholenden Abfolge



zuerst den Cache zu Primen, dann das Opferprogramm rechnen zu lassen und anschließend zu Proben. In der Priming-Phase werden mittels der Eviction-Set gezielt Cache-Sets vollständig mit den Daten aus dem Eviction-Set belegt. In der anschließenden Berechnungsphase werden einige Einträge aus den geprimten Cache-Sets vom Opferprogramm verdrängt. Abschließend berechnet die Angreiferin die Summe der Zugriffszeiten auf alle Einträge in einem Eviction-Set. Sofern das Opferprogramm in dem zum Eviction-Set korrelierenden Cache-Set Einträge verdrängt hat, kann die Angreiferin eine Abweichung nach oben in ihrer Messung feststellen, da die verdrängten Einträge eine erhöhte Zugriffszeit beisteuern.

Die zur Durchführung essenziellen Eviction-Sets sind nicht immer leicht zu finden, da die virtuellen Adressen in manchen Umgebungen nur eingeschränkt zugänglich sind. So lässt sich häufig nur garantieren, dass maximal die untersten 12 virtuellen Adressbits mit den physikalischen Adressenbits identisch sind, da die typische Page-Size  $4096(2^{12})$  Bytes beträgt. In solchen Fällen müssen die Eviction-Sets in einem Trial-and-Error Verfahren ermittelt werden, wie es der Algorithmus beschreibt.

#### 1.3.3 RSA

Falls RSA angegriffen wird TODO

RSA Key Gen, RSA Verschlüsselung Entschlüsselung

---

**Algorithm 1:** Psuedo-Code für Eviction-Set Algorithmus

---

```

1 Function EvictionSetFinder(memoryBlocks)
2   groups  $\leftarrow$  empty while size(memoryBlocks) > 0 do
3     evictionSet  $\leftarrow$  empty
4     witness  $\leftarrow$  expand(evictionSet, memoryBlocks)
5     if witness  $\neq$  failed then
6       contract(evictionSet, memoryBlocks, witness)
7       witnessSet  $\leftarrow$  collect(evictionSet, memoryBlocks, witnessSet)
8       groups.add(union(evictionSet, witness, witnessSet))
9
9 Function Expand(evictionSet, memoryBlocks)
10  while size(candidateSet) > 0 do
11    witnesss = SelectRandomItem(candidateSet)
12    if checkevict(evictionSet, witnesss) then
13      return witnesss
14    evictionSet.add(witnesss)
15  return failed;
16
16 Function Contract(evictionSet, memoryBlocks, witness)
17  foreach candidate in evictionSet do
18    if checkevict(evictionSet, witness) then
19      mermoryBlocks.add(candidate)
20      evictionSet.add(candidate)
21
21 Function Collect(evictionSet, memoryBlocks)
22  witnessSet = empty
23  foreach candidate in mermoryBlocks do
24    if checkevict(evictionSet, candidate) then
25      memoryBlocks.delete(candidate)
26      witnessSet.add(candidate)
27  return witnessSet;

```

---

## 2 Implementierung

Das folgende Kapitel beschreibt mit Hilfe welcher Softwaretools der Cache-Angriff implementiert wird.

Um das allgemeinere und praxisnähere Angriffsmodell umsetzen und den Angriff schon durch den Besuch einer Website zu starten, liegt der komplette Angriffscode in JavaScript und Webassembly vor. Frühere Implementierungen von Cache-Angriffen im Browser [5] hatten noch keine Möglichkeit Webassembly zu verwenden, weshalb diese den kompletten Angriffscode in JavaScript schreiben mussten. Webassembly ermöglicht hardwarenähere Programmierung und dem Vorteil, dass Code nicht wie in JavaScript während Laufzeit optimiert wird. Des Weiteren steht mit dem emscripten-Compiler ein Tool bereit, welches die Übersetzung von C-Code in Webassembly anbietet. Somit kann bestehender Angriffscode in C, in diesem Fall von Mastik, als Grundlage verwendet werden, und eine komplette fehleranfällige Neuimplementierung in JavaScript entfällt.

### 2.1 Timer in JavaScript

Der hier ausgeführte Cache-Angriff benötigt präzise Timer, welche eine Auflösung von unter  $x$  ns bereitstellen sollten. Wie im Diagramm zu sehen (TODO diagram einfügen) ist Zugriffszeit einem Cache-Miss im Mittel  $x$  ns höher als bei einem Cache-Hit. Das heißt bei einer Auflösung von  $x$  ns lässt sich nicht mehr in allen Fällen ein Miss von einem Hit unterscheiden. Dennoch lässt sich die Suche nach Eviction-Sets auch mit schlechteren Timerauflösungen bewerkstelligen, indem Operationen mehrfach ausgeführt werden und die Differenz der aufsummierten Zeiten zur Bewertung herangezogen wird. Im Eviction-Set Algorithmus könnte etwa die Funktion *checkevict* wie in 2 angepasst werden, wobei der Parameter *repeatIterations* abhängig von Timerauflösung gewählt wird. Weiter besteht jedoch das Problem schwache Aktivitäten im Cache-Set während des eigentlichen

## 2 Implementierung

Angriffs aufzudecken, da im Worst-Case nur ein Eintrag aus dem beobachteten Cache-Set verdrängt wird und somit lediglich die Zugriffszeit zwischen einem Hit und Miss ausschlaggebend ist. In diesem Fall könnte die Dauer mehrerer Prime+Probe Iterationen gesamtheitlich gemessen werden, und zwar in der Vermutung, dass auf die für die Verdrängung verantwortliche Adresse über die Zeit mehrfach zugegriffen wird. Die direkten Auswirkungen eines niedrig aufgelösten Timers sind also eine geringere zeitliche Auflösung oder die Nichtregistrierung von schwachen Cache-Aktivitäten.

---

**Algorithm 2:** Psuedo-Code für *checkevict* im Fall von einer niedrig aufgelösten getTimestamp

---

```
1 Function checkevict(possibleEvictionSet, witness)
2   timestampBefore <- getTimestamp()
3   for i=1 to repeatIterations do
4     |   accessMemory(possibleEvictionSet)
5     |   accessMemory(witness)
6   timestampAfter <- getTimestamp()
7   return timestampAfter - timestampBefore > threshold;
```

---

Der W3C hat die High Resolution Time API spezifiziert, welche die Methode `performance.now()` beinhaltet, die einen aktuellen Timestamp zurückgibt. Im Firefox hatte die Methode in früheren Versionen eine hinreichend genaue Auflösung im Nanosekundenbereich, wobei in Reaktion auf die Sicherheitslücken Meltdown und Spectre die Auflösung schrittweise auf 2ms im aktuellen Firefox 60 abgesenkt wurde. Auch in den Browsern Edge und Chrome wurden im Zuge der Veröffentlichung von Meltdown und Spectre die Auflösung von `performance.now()` verringert, allerdings wird auf den zurückgegebenen Timestamp zusätzlich ein Timerjitter addiert. So bieten Edge und Chrome zurzeit (Stand Juni 18) eine Auflösung von 20us + 20us Jitter respektive 100us + 100us Jitter. Das Paper "Fantastic Timers and where to find them" beschreibt diverse andere Methoden um mit Hilfe von JavaScript Timer zu generieren. Allerdings ist nur eine geeignete Methode dabei, da die Auflösung aller anderen mindestens im hohen einstelligen us Bereich liegt und somit der Parameter *repeatIterations* auf Werte oberhalb von TODO x gesetzt werden müsste, um zuverlässig Eviction Sets zu finden. Hierdurch würde die benötigte Ausführungszeit zum Finden der Eviction Sets auf ein Maß ansteigen, welches dann nicht mehr

zum angenommenen Angriffsmodell passen würde.

### 2.1.1 Verdeckter Kanal

Die maximale Sendegeschwindigkeit eines Kanals ist durch die Rate mit welcher der Sender ein beliebiges Cache-Set primen kann, begrenzt. Damit der Empfänger ein zufälliges Rauschen von einem Priming unterscheiden kann, sollte der Sender mehrere Einträge aus dem zu primenden Cache-Set verdrängen, wobei im Optimalfall die Anzahl der zugegriffenen Speicheradressen der Assoziativität des Caches entspricht. Hiermit wird die Wahrscheinlichkeit erhöht, dass sich die vom Empfänger im Probe-Schritt gemessene Zugriffszeit signifikant von Fällen unterscheidet in denen zufällig einzelne Einträge aus dem überwachten Cache-Set verdrängt werden. Im Folgenden sollen verschiedene Methoden verglichen werden um ein Cache-Set zu primen, indem entweder die Anzahl der zugegriffenen Speicheradressen oder die Zugriffsmethode verändert wird. Wenn etwa die Anzahl der zugegriffenen Speicheradressen verringert wird, sind auf der einen Seite mehr Timeslots in einem Zeitabschnitt möglich und die Chance, dass benachbarte Timeslots zusätzlich beeinflusst werden, sinkt. Auf der anderen Seite sind die messbaren Ausschläge der Zugriffszeiten verringert, wodurch ein bewusst geprimtes Cache-Set schwieriger von einem Messrauschen oder zufälligen Zugriffen unterschieden werden kann. Sende- und Empfangsseite können durchaus abweichende Parameter verwenden, wenn wie etwa im vorliegenden Fall der Empfänger langsamer als der Sender arbeitet. Um die Timeslots anzugleichen, könnte der Empfänger die Dauer für eine Priming-Operation durch die Senkung der Anzahl der zugegriffenen Speicheradressen verringern und der Empfänger andersherum die Dauer für eine Priming-Operation erhöhen.

---

#### Algorithm 3: Psuedo-Code für Pointer-Chasing-Methode

---

```

1 Function AccessTimeEvictionSet(pointerToAddress)
2   pointerToAddressFirst  $\leftarrow$  pointerToAddress
3   timestampBefore  $\leftarrow$  getTimeStamp()
4   while pointerToAddressFirst  $\neq$  pointerToAddress do
5     | pointerToAddress  $\leftarrow$  readValue(pointerToAddress)
6   return getTimeStamp() - timestampBefore

```

---

## 2 Implementierung

Auf dem Testrechner benötigt ein in C geschriebenes Sendeprogramm für eine Million Prime-Vorgänge mit 16 Adressen und der Single-Pointer-Chasing-Methode (siehe Algorithmus 3) etwa 323 Millionen Taktzyklen. Im Optimalfall kann im Timeslot  $x$  ein erfolgreicher Prime-Vorgang als 1 und ein nicht erfolgreicher Prime-Vorgang als 0 interpretiert werden. Bei einem typischen All-Core Turbo Takt von 3,4 Ghz des i7-4770 ergibt sich so eine maximale Senderate von 10,5 Mbit/s. Diese Rate wird jedoch vom Empfänger beschränkt, welcher zusätzlich noch eine Zeitmessung durchführen muss. Der Worst-Case ist hier ein in Webassembly geschriebener Empfänger, da dort eine Zeitmessung kostenintensiver ist. In Chromium 66 können eine Million Messungen eines Cache-Sets in etwa 200ms durchgeführt werden. Im Mittel dauert eine Messung also 0,2s, womit ein Empfangsrate von maximal 5 MBit/s realisiert werden kann.

Im Folgenden soll die maximal mögliche Senderate unter optimalen Bedingungen ermittelt werden. Hierfür wird im Voraus ein Cache-Set ausgewählt auf dem im Idle-Zustand des Systems ein geringes Rauschen herrscht. Um die Synchronisation des Senders und Empfängers aufrechtzuerhalten, wird nach 10 gesendeten Bits ein Synchronisationsblock eingefügt, welcher durch  $sb$  Prime-Vorgänge auf der Senderseite erzeugt wird. Eine 1 wird durch  $s$  Prime-Vorgänge repräsentiert und eine 0 durch das Unterlassen der Prime-Vorgänge. Um die einzelnen Bits auseinanderzuhalten wird zwischen jedem gesendeten Bit eine Pause von  $p$  Taktzyklen eingelegt.

### 2.1.2 Angriffe auf RSA Key Generierung

Details zu Implementierung in Mozilla NSS

Was wird angegriffen

Verweis auf Paper Cache-Timing Attacks on RSA Key Generation

## Literaturverzeichnis

- [1] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.
- [2] Ian Cutress. The amd zen and ryzen 7 review: A deep dive on 1800x, 1700x and 1700. <https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700/9>, 2017. Accessed: 2018-06-14.
- [3] Ian Cutress. Intel announces skylake-x: Bringing 18-core hcc silicon to consumers for \$1999. <https://www.anandtech.com/show/11464/intel-announces-skylakex-bringing-18core-hcc-silicon-to-consumers-for-1999/3>, 2017. Accessed: 2018-06-14.
- [4] Intel. Intel® 64 and ia-32 architectures optimization reference manual. <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>, 2016. Accessed: 2018-06-14.
- [5] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 1406–1418, New York, NY, USA, 2015. ACM.
- [6] PassMark. Amd vs intel market share. [https://www.cpubenchmark.net/market\\_share.html](https://www.cpubenchmark.net/market_share.html), 2018. Accessed: 2018-06-16.
- [7] Colin Percival. Cache missing for fun and profit. 2005.
- [8] Henry Wong. Intel Ivy Bridge cache replacement policy, jan 2013. Accessed: 2018-06-16.